DOI 0.1109/TCAD.2018.2878187

Support-Reducing Decomposition for FPGA Mapping

Lucas Machado and Jordi Cortadella, *Fellow, IEEE*

***Abstract*—Decomposition is a technology-independent process, in which a large complex function is broken into smaller, less complex functions. The costs of two-level or factored-form representations (cubes and literals) are used in most decomposition methods, as they have a high correlation with the area of cell-based designs. However, this correlation is weaker for field-programmable gate arrays (FPGAs) based on look-up tables. Furthermore, local optimizations have limited power due to the structural bias of the circuit descriptions.**

This paper tries to reduce the structural biasing by remapping the LUT network and decomposing the derived functions using the support as cost function. The proposed method improves the FPGA mapping results of a commercial tool for the 20 largest MCNC benchmarks, with gains of 28% in delay plus 18% in area when targeting delay, and a reduction of 28% in area plus 14% in delay with area as cost function. Results with 23% less area and 6% less delay are obtained after physical synthesis (post place-and-route). Moreover, 12 of the best known results for delay (and 3 for area) of the EPFL benchmarks are improved.

***Index Terms*—Logic decomposition, logic synthesis, performance optimization, support reducing, technology mapping.**

I. Introduction

FIELD-PROGRAMMABLE gate arrays (FPGAs) are integrated circuits consisting of programmable logic blocks and interconnections. FPGAs can be reprogrammed multiple times, and have a much smaller initial cost and production time in comparison with application-specific integrated circuits (ASIC). For these reasons, FPGAs are largely used for ASIC prototyping and low-volume applications. However, when compared to ASICs, the flexibility given by FPGAs comes at the expense of larger area, power consumption, and delay [2]. Recently, FPGAs started to be employed in the optimization of specific tasks in data centers, with technology leaders making efforts in hybrid solutions with ASICs and FPGAs [3], [4].

The FPGA implementation process inherited many techniques from the ASIC design flow. The use of well-established methods enabled the fast growing and wide usage of FPGAs, but these algorithms generally have cost functions customized for cell-based designs, in which the area is proportional to the number of transistors. Usual cost functions in logic synthesis are cubes in sum-of-product (SOP) forms, literals in Boolean function expressions, or nodes and levels of And-Inverter Graphs (AIGs). On the other hand, FPGAs based on look-up tables (LUTs) are composed of logic blocks with k inputs (typically 4 to 6), and each LUT can implement any logic function of up to k inputs. A study on this miscorrelation is presented in [5], showing that the reduction of nodes and levels in AIGs does not necessarily translates to fewer LUTs or less logic depth in the FPGA mapping generated.

This work was performed with the support of CNPq, Conselho Nacional de Desenvolvimento Científico e Tecnológico, Brasil, in part by the Spanish Ministry for Economy and Competitiveness and the European Union (FEDER funds) under grant TIN2017-86727-C2-1-R, and in part by the Generalitat de Catalunya (2017 SGR 786). This paper was recommended by Associate Editor R. Drechsler. (*Corresponding author: Lucas Machado.*)

The authors are with the Computer Science Department, Universitat Politècnica de Catalunya, 08034 Barcelona, Spain (e-mail: lmachado@cs.upc.edu; jordi.cortadella@upc.edu).

A. Previous work

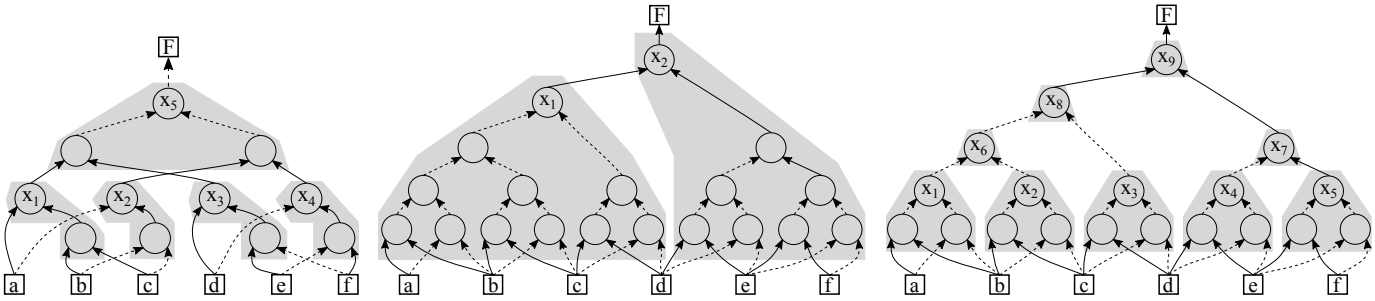
Several works on FPGA mapping are based on cut-enumeration, performing a covering of the subject graph using k-cuts [6]–[8]. These cut-based techniques vary on the algorithms, parameters, and cost functions used for the cut-enumeration and covering. Nevertheless, the quality of the solution heavily depends on the structure of the subject graph.

A second group of works rely on *Binary Decision Diagrams* (BDDs) to perform FPGA mapping [9]–[12]. BDDs usually provide *per se* a good starting point for FPGA mapping, as the redundant variables are removed and the structure size is reduced. Also, BDDs enable the use of functional techniques, reducing the structural bias. However, the complexity of BDDs increases significantly with the number of variables, becoming computationally unfeasible for large designs. Thus, BDD-based methods are often applied to portions of the circuit (partial collapsing), but these methods are also structurally biased. This work proposes to combine these two strategies, using both functional decomposition and cut-based mapping.

The idea of performing decomposition while reducing the support (and targeting FPGAs) has already been proposed. The support is minimized using don't cares in [13], as explained in [14]. In [15], it is proposed a complex decomposition aiming support minimization, by identifying the compatibility of all variables (or classes) in the bound-set. BoolMap [9] uses the decomposition proposed in [15]. Our work proposes the restructuring of the LUT network using the support size as cost function, with the aid of simple and fast decompositions.

The support-reducing techniques presented in this paper are well-known methods, with the exception of the abstraction-based decompositions (see Section IV-E). Other decomposition methods could be considered, such as [15]–[18], which are slower, but could improve the quality of results. Still, the *key idea* is to consider the support size as the cost function for decomposition, which restructures the subject graph targeting LUT-based FPGAs, and not the techniques incorporated.

In this paper, a decomposition of a function F is considered *support-reducing* if the decomposing functions have their support size smaller than F. This definition differs from [19], which limits the term *support-reducing* to disjoint-support decompositions (DSD).



(a) Factorization (LUT4 mapping) (b) Support-reducing decomposition (LUT4 mapping) (c) Support-reducing decomposition (LUT2 mapping)

Fig. 1. Functionally equivalent, but structurally different AIGs, obtained via (a) algebraic factorization, and (b)(c) support-reducing decomposition.

B. Contributions of this paper

This work proposes two main contributions:

1) A functional decomposition, which is guided by the size of the support, and it is based on simple and fast support-reducing techniques.
2) A recursive remapping approach, that reduces the structural bias of the subject graph, and uses the FPGA mapping metrics as cost function.

The proposed approaches are implemented into an FPGA remapping tool, named **S**upport-**R**educing Re**map**ping (**SR-map**). By remapping the results of a commercial tool for the 20 largest MCNC benchmarks, SR-map is able to reduce delay in 28% (plus 18% in area) when targeting delay, and improve area in 28% (plus 14% in delay) with area as cost function. The main reasons for these improvements are:

1) The FPGA mapping metrics are used to guide the resynthesis algorithm, instead of literals and cubes.
2) A new and aggressive collapsing strategy is applied, instead of a local *partial collapsing*.
3) A *new structure* is generated by a support-reducing functional decomposition.

The goal of using the mapping result as cost function is to reduce the miscorrelation between intermediate and final results, accepting transformations that will contribute to improve the final solution [5]. This is possible with fast and high-quality FPGA mapping algorithms [8].

BDD-based methods often rely on the partial collapsing of the subject graph [9]–[11]. The effectiveness of this process depends on the structure of the subject graph, which can easily reach a local minimum. This work performs a recursive global collapsing on the LUT network (see Section V), with the goal of obtaining a result less biased by the structure of the subject graph.

The support size as cost function in the decomposition makes sense for FPGAs: a k-input function with *any* number of literals can be implemented with a single LUT of k inputs. This concept is illustrated with an example in Section II.

The rest of the paper is organized as follows. Some preliminary concepts are described in Section III. Section IV presents the support-reducing decomposition, and the recursive remapping is explained in Section V. Section VI provides the results and comparisons with academic tools and a commercial tool. Section VII concludes the paper.

TABLE I
COMPARISON OF THE FPGA MAPPING FOR THE AIGS OBTAINED VIA ALGEBRAIC FACTORIZATION AND SUPPORT-REDUCING DECOMPOSITION.

LUT size	Algebraic factorization		Support-reducing decomposition	
	LUTs	Levels	LUTs	Levels
2 inputs	11	4	9	4
3 inputs	6	3	4	2
4 inputs	5	2	2	2
5 inputs	3	2	2	2

II. MOTIVATING EXAMPLE

In this section, an example is presented to illustrate the support-reducing decomposition. Let us assume the following expression, which represents a 6-input Boolean function:

$$F(a,b,c,d,e,f) = abcde\bar{f} + \bar{a}\bar{b}\bar{c}\bar{d}\bar{e}f \quad (1)$$

This expression has 12 literals, and it is also the optimal AND/OR factored form, as there is no other expression with fewer literals. The AIG shown in Fig. 1(a) is derived from the expression in (1). A structural FPGA mapping targeting LUTs with 4 inputs is also shown in Fig. 1(a), with the 5 shadowed regions representing the LUT covering of the AIG. The FPGA mapping derived by [8] is the following:

$$x_1 = abc,\ x_2 = de\bar{f},\ x_3 = \bar{a}\bar{b}\bar{c},\ x_4 = \bar{d}\bar{e}f$$
$$x_5 = x_1x_2 + x_3x_4$$

By applying the proposed support-reducing decomposition on (1), the following expression (with 20 literals) is obtained:

$$F = (ab + \bar{a}\bar{b})(bc + \bar{b}\bar{c})(cd + \bar{c}\bar{d})(de + \bar{d}\bar{e})\overline{(ef + \bar{e}\bar{f})} \quad (2)$$

In this case, the function was decomposed with the abstraction-based AND bi-decomposition (see Section IV-E). At each step, a variable is removed from the support of the decomposed function, with the aid of the existential abstraction $\exists x_i F$. The function is decomposed in the form $F = (\exists x_i F) \cdot H$, with H obtained via don't care minimization.

The AIG presented in Fig. 1(b) is derived from expression (2). This AIG has 8 more nodes than the one in Fig. 1(a). This means that it would likely result in a circuit with larger area, if implemented as a cell-based design. However, its mapping with 4-input LUTs has only 2 LUTs, whereas the one for Fig. 1(a) has 5. The covering given by [8] is the following:

$$x_1 = (ab + \bar{a}\bar{b})(bc + \bar{b}\bar{c})(cd + \bar{c}\bar{d})$$
$$x_2 = x_1(de + \bar{d}\bar{e})\overline{(ef + \bar{e}\bar{f})}$$

Similarly, the FPGA mapping for LUTs of different sizes would require fewer LUTs for the AIG obtained by the support-reducing decomposition, as shown in Table I. The derived FPGA mapping is smaller for all cases, even for 2-input LUTs, which is illustrated in Fig. 1(c).

Note that a and $\bar{a}$ are different literals. This makes sense for cell-based designs, as each literal will result in a transistor. Still, a and $\bar{a}$ are the same in terms of support, as both refer to the variable a. Therefore, the *support* is the key cost function for the proposed decomposition, generating a structure more suitable for LUT-based FPGAs, even with larger AIGs.

III. Preliminaries

A. Boolean Functions

An incompletely specified Boolean function (ISF) $F(X)$ of n Boolean variables is a mapping from an n-dimensional into a 1-dimensional Boolean space: $\{0,1\}^n \rightarrow \{0,1,-\}$, where '$-$' denotes a *don't care* value. The sub-domains of F that evaluate to '1', '0' and '$-$' are the ON-set, OFF-set and DC-set, respectively. F is a completely specified function if its DC-set is empty. The set $X = (x_1, x_2, \ldots, x_n)$ is denoted as the *support* of F, and $|F|$ denotes the size of X (n variables). The complement of F is denoted as $\overline{F}$. The logic operations AND, OR and XOR are denoted as '$\cdot$' (or simply ' '), '$+$', and '$\oplus$', respectively.

B. Cofactors and derivations

The positive (negative) *cofactor* operation of $F(X)$ with respect to the variable $x_i \in X$ consists of assigning x_i to one (zero) in $F(X)$, which can be represented as F_{x_i} ($F_{\overline{x_i}}$). A *cube-cofactor* consists of performing the cofactor operation recursively, e.g., assigning the variables $\{x_i, x_j\} \subseteq X$ in $F(X)$ to $x_i = 0$ and $x_j = 1$, which can be denoted as $F_{\overline{x_i}x_j}$.

Cofactors can be used to extract information from F with respect to a variable in its support. Typical cofactor derivations are: the Boolean difference (3), the existential (4) and the universal abstractions (5), the *Shannon expansion* (6), and the positive (7) and negative (8) *Davio expansions*.

$$\delta F/\delta x_i = F_{\overline{x_i}} \oplus F_{x_i} \tag{3}$$

$$\exists x_i F = F_{\overline{x_i}} + F_{x_i} \tag{4}$$

$$\forall x_i F = F_{\overline{x_i}} \cdot F_{x_i} \tag{5}$$

$$F = \overline{x_i} \cdot F_{\overline{x_i}} + x_i \cdot F_{x_i} \tag{6}$$

$$F = (x_i \cdot \delta F/\delta x_i) \oplus F_{\overline{x_i}} \tag{7}$$

$$F = (\overline{x_i} \cdot \delta F/\delta x_i) \oplus F_{x_i} \tag{8}$$

C. And-Inverter Graphs

An *And-Inverter Graph* (AIG) is a directed-acyclic graph (DAG) in which each node has 0 incoming edges - the *primary inputs* (PIs), or 2 incoming edges - the *AND nodes*, or 1 incoming edge - the *primary outputs* (POs). Each edge can be negated or not. Sequential elements are considered as PI/PO pairs. Examples of AIG are shown in Fig. 1: the dashed lines are negated edges, the circles are AND nodes, the squares at the bottom are PIs, and the squares at the top are POs.

AIGs have a high correlation with its derived circuit implementation, with the area correlated with the size of the graph, and the delay proportional to the number of levels between PIs and POs. *Structural hashing* [20] is an operation which ensures that the AIG has only one AND node with the same incoming edges, considering permutation. Balancing [21], rewriting [22], refactoring [20] and resubstitution [23] are AIG transformations applied to reduce the nodes and levels of AIGs. Some commonly used scripts in ABC [24] are *dc2* and *compress2rs* [5], [25], which iterate these transformations.

D. FPGA mapping

Technology mapping consists of transforming a technology-independent *subject graph* into a network of gates from a technology. In FPGAs, the technology typically consists of LUTs, which are logic blocks that can be configured to implement any logic function of up to k variables.

In ABC [24], the subject graph is a structurally hashed AIG, and the FPGA mapping is performed on top of this structure [8]. The FPGA mapping may vary significantly for functionally equivalent but structurally different AIGs [26]. For example, the AIGs shown in Fig. 1(a) and Fig. 1(b) implement the same function. However, the FPGA mapping (k=4) for the AIG of Fig. 1(a) has 5 LUTs, whereas the mapping for the AIG in Fig. 1(b) has 2 LUTs.

E. Binary Decision Diagrams

A *Binary Decision Diagram* (BDD) is a well-known representation of Boolean functions [9]–[11]. A BDD is a DAG with two terminal nodes (0 and 1), and each nonterminal node represents a Boolean variable with two outgoing edges: the 0-edge and the 1-edge. A reduced-ordered BDD (ROBDD) is a BDD in which the nonterminal nodes are organized in a fixed variable order, in such a way that the number of BDD nodes is reduced, and the redundant variables are removed. Notice that ROBDDs are also a representation of the Shannon expansion, which is a support-reducing decomposition.

In this work, ROBDDs are referred as BDDs, and are the representation of choice for the support-reducing decomposition. BDDs are an efficient representation (with a few exceptions) and are able to handle a larger amount of variables than other functional representations, e.g., truth tables. Also, there are modern software libraries which can be used to implement efficiently the techniques presented in Section IV.

F. Collapsing

In this work, the process of *collapsing* a circuit [21] is performed for each output individually. The result of the collapsing process is the logic function of a primary output based on the primary inputs, as shown in the example of Fig. 2. The output function obtained is the same regardless of the circuit structure, therefore the structural don't cares are removed [27]. Notice that logic sharing between outputs is potentially lost in this process, as observed in Fig. 2. This approach may result in larger area, but also increases the possibilities of reducing circuit delay.

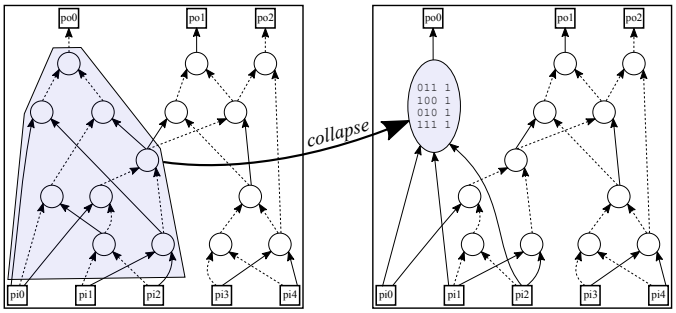


Fig. 2. Example of the collapsing process for a primary output.

In order to obtain an AIG from a collapsed function, and therefore a subject graph for FPGA mapping, it is necessary to decompose such function. For example, the decomposition can be performed via algebraic factorization [28] (*strash* command in ABC), or using the BDD structure, replacing each BDD node by a multiplexer (*muxes* command in ABC). In this work, the support-reducing decomposition is applied (see Section IV), followed by structural hashing.

IV. Support-reducing functional decomposition

This section presents the proposed functional decomposition, which is based on simple and fast support-reducing techniques. The decomposition is technology-independent, i.e., it is agnostic to the target FPGA technology. The goal is to generate a structure guided by the support size. The resulting subject graph typically produces a faster or smaller LUT network, but there are corner cases that poor results are obtained, e.g., multiplier. For this reason, the remapping approach in Section V selects the best result between the existing LUT network and the one derived by the decomposition.

The input for the decomposition is an ISF. It is possible to identify external don't care conditions (*DC*) and use them as input to the decomposition. However, in this work, the only don't care conditions identified are from the internal recursions of the method. The output of the decomposition is a Boolean network consisting of logic gates from the set {*AND2, OR2, XOR2, MUX21, AO21, AX21*}, which are required to implement the techniques considered.

Fig. 3 presents a pseudo-code of the algorithm. The trivial cases are checked at line 4. If the DC-set is not empty, then minimization is applied (line 7), updating F if the minimized function has a support smaller or equal to F. The minimization can be implemented by any method that accepts an ISF, e.g., Espresso [28], BDD minimization [29], [30].

The decomposition method described in Fig. 4 is invoked at line 12, which receives an ISF as input and returns a solution consisting of a decomposing gate (*op*) and a set of functions ($\langle F_1, ..., F_n \rangle$). If the solution has no disjoint support, then the satisfiability don't care (SDC) conditions are calculated by CALCSDC (line 15), which is implemented as in [27].

Each derived function is decomposed recursively, generating a Boolean network (line 17). The network obtained is connected to the related input in the decomposing gate (*op*) at line 18. Notice that the resulting network is also a tree, and

```
1: function SUPPORTREDUCEDECOMPOSITION(F, DC)
   Input: An ISF, with the ON-set (F) and the DC-set (DC)
   Output: A Boolean network (tree) that implements the ISF
2:   // check for trivial cases (constants, variables)
3:   // the support size of F is denoted as |F|
4:   if |F| ≤ 1 then return F
5:   // if DC-set is not empty, then minimize F
6:   if DC ≠ ∅ then
7:     F_min = MINIMIZE(F, DC) // Espresso [28], BDD reduction [29]
8:     // accept F_min if support is reduced or the same
9:     if |F_min| ≤ |F| then F = F_min
10:  // perform decomposition
11:  // op ∈ {AND2, OR2, XOR2, MUX21, AO21, AX21}
12:  (op, ⟨F_1, ..., F_i, ..., F_n⟩) = DECOMPOSEFUNCTION(F, DC)
13:  for each function F_i in ⟨F_1, ..., F_i, ..., F_n⟩ do
14:    // calculate satisfiability don't care as in [27]
15:    DC_i = DC + CALCSDC(op, i, ⟨F_1, ..., F_i, ..., F_n⟩)
16:    // decompose F_i recursively
17:    network = SUPPORTREDUCEDECOMPOSITION(F_i, DC_i)
18:    op.connect(i, network)          // connect network to gate input i
19:  return op                                          // root of the tree
```

Fig. 3. Pseudo-code of the proposed support-reducing decomposition.

```
1: function DECOMPOSEFUNCTION(F, DC)
   Input: An ISF, with the ON-set (F) and the DC-set (DC)
   Output: A decomposition (op, ⟨F_1, ..., F_i, ..., F_n⟩)
2:   Q = ∅                  // priority queue of potential solutions
3:   // check for essential literals
4:   DECOMPOSEESSENTIALS(F, 1, Q)
5:   DECOMPOSEESSENTIALS(F̄, 0, Q)
6:   // if essential literals found, return
7:   if Q ≠ ∅ then return best solution ∈ Q
8:   // check one-variable decompositions
9:   DECOMPOSEONEVARIABLE(F, DC, Q)
10:  // check two-variable decompositions
11:  DECOMPOSETWOVARIABLES(F, DC, Q)
12:  return a solution with the lowest cost ∈ Q
```

Fig. 4. Pseudo-code for an step of the support-reducing decomposition.

the task of sharing logic is postponed to structural hashing and AIG optimizations (see Section V).

The method in Fig. 4 performs several support-reducing techniques on the input function F, selecting the one with the lowest sum of support sizes, given that all functions in $\langle F_1, ..., F_n \rangle$ have a support size smaller than $|F|$. If several solutions are found, additional costs are considered (see Section IV-A). Other techniques could be incorporated [15]–[18], which are slower but could improve the results. Still, the idea is to use simple and fast techniques that reduce the support, obtaining an efficient method that is able to produce good results by using the support size as cost function.

The support-reducing techniques tried are: (1) essential literals (lines 4-5), which is a *simple* and fast disjoint-support decomposition (DSD); (2) trying to remove one variable from the support (line 9), using Shannon and Davio expansion (and its simplifications); (3) trying to remove two variables (line 11), with additional DSD techniques; and (4) a new bi-decomposition method, based on the universal and existential abstractions, applied to one and two variables.

A. Cost function

In this work, the cost function is the sum of support sizes of the derived functions, i.e., $\min \sum_{i=1}^{i=n} |F_i|$. Moreover, a solution is only accepted if all derived functions have a support size smaller than F, i.e., $\forall_i |F_i| < |F|$. Additionally, if there is more than one solution with the smallest sum of support sizes, then the following costs are considered, in this order:

```
1: procedure DECOMPOSEESSENTIALS(F, P, Q)
   Input: Boolean function F, polarity P, priority queue of decompositions Q
   Post: decompositions added to Q
2:     E = ⟨e_1, ..., e_n⟩                          // set of n essential literals of F
3:     if E == ∅ then return
4:     H = F_{e_1...e_n}                            // cube-cofactor of F w.r.t. E
5:     if H ≠ 1 then
6:         G = (e_1 · ... · e_n)                    // AND of all essential literals
7:         // polarity P ∈ {0, 1}
8:         if P == 1 then Q.add(G · H)              // AND2
9:         else Q.add(G' + H)                       // OR2
10:    else
11:        G_1 = (e_1 · ... · e_{n/2})              // AND of essential literals 1 to n/2
12:        G_2 = (e_{n/2+1} · ... · e_n)            // AND of essential literals n/2+1 to n
13:        if P == 1 then Q.add(G_1 · G_2)          // AND2
14:        else Q.add(G_1' + G_2')                  // OR2
```

Fig. 5. Pseudo-code for decomposition using essential literals.

1) The sum of squares of the BDD sizes [31], targeting a balanced solution, which favors delay reduction.
2) The gate implementation cost in CMOS transistors, e.g., an AND2 gate costs less than a MUX21 or an XOR2.

As BDDs are the representation of choice, the cost function exploits their structure to guide the decomposition, but similar costs could be derived for other representations. For example, the support size could be used instead of the BDD size.

B. Essential literals

Fig. 5 describes the decomposition method using *essential literals*, i.e., literals that are common to all prime implicants. For example, given $F(X)$ and $\{a, b, c\} \subseteq X$, if $\{a, b, \bar{c}\}$ are essential literals of $F(X)$, then F can be rewritten as $F(X) = (ab\bar{c})F_{ab\bar{c}}$. Similarly, given $G(X) = \overline{F(X)}$ and $\{x, y, z\} \subseteq X$, if $\{\bar{x}, y, z\}$ are essential literals of $G(X)$, then F can be decomposed as $F(X) = \overline{(\bar{x}yz)} + \overline{G_{\bar{x}yz}}$.

Decomposition with essential literals is checked first and preferred to the other techniques, as it is a fast DSD method which removes the *simple* part of the decomposition. The essential literals of F are checked at line 8 (polarity $P = 1$), and the one for $\overline{F}$ at line 9 (polarity $P = 0$). If the function is solely composed of essential literals, i.e., the cube-cofactor w.r.t. to the essential literals is the constant 1 (F is a cube), then a balanced decomposition is performed (lines 13-14).

Example: Consider the function $F = \bar{a}\bar{c}(b(d + \bar{f}) + \bar{e})$, which has the essential literals $\{\bar{a}, \bar{c}\}$. By calculating the cube cofactor $F_{\bar{a}\bar{c}} = (b(d + \bar{f}) + \bar{e})$, it is possible to decompose the function $F = (\bar{a}\bar{c})(b(d + \bar{f}) + \bar{e})$. Let us consider another function $G = (ab + cd) + (e + f)$, which has no essential literals. The complement function $\overline{G} = H = (\bar{a} + \bar{b})(\bar{c} + \bar{d})\bar{e}\bar{f}$ has the essential literals $\{\bar{e}, \bar{f}\}$. The cube cofactor in this case is $H_{\bar{e}\bar{f}} = (\bar{a} + \bar{b})(\bar{c} + \bar{d})$, deriving the decomposition $G = \overline{(\bar{e}\bar{f})} + \overline{(\bar{a} + \bar{b})(\bar{c} + \bar{d})} = (e + f) + (ab + cd)$

C. One-variable decompositions

The basic one-variable support-reducing decompositions are given by the Shannon expansion (6), and the Davio expansions (7-8). These methods isolate one variable, thus reducing the support size of the derived functions in at least one. Simplifications of these expansions can be obtained given specific conditions, as shown in (9). Essential literals cover the cases in which one of the cofactors is a constant.

```
1: procedure DECOMPOSEONEVARIABLE(F, DC, Q)
   Input: ON-set (F), DC-set (DC), priority queue of decompositions Q
   Post: decompositions added to Q
2:     for each variable x_i ∈ support of F do
3:         if δF/δx_i == 1 then
4:             Q.add(x_i ⊕ F_{x_i'})
5:             Q.add(x_i' ⊕ F_{x_i})
6:             return
7:         if ∃x_i F == F_{x_i'} then
8:             Q.add((x_i' · F_{x_i'}) + F_{x_i})                    // AO21
9:         else if ∃x_i F == F_{x_i} then
10:            Q.add((x_i · F_{x_i}) + F_{x_i'})                     // AO21
11:        else // full Davio and Shannon expansions
12:            Q.add((x_i · δF/δx_i) ⊕ F_{x_i'})                     // AX21
13:            Q.add((x_i' · δF/δx_i) ⊕ F_{x_i})                     // AX21
14:            Q.add(x_i' · F_{x_i'} + x · F_{x_i})                  // MUX21
15:        // one-variable abstraction-based bi-decompositions
16:        if ∃x_i F ≠ 1 then
17:            G = ∃x_i F
18:            H = MINIMIZE(F, G' + DC) // Espresso [28], BDD reduction [29]
19:            Q.add(G · H)                                          // AND2
20:        if ∀x_i F ≠ 0 then
21:            G = ∀x_i F
22:            H = MINIMIZE(F, G + DC) // Espresso [28], BDD reduction [29]
23:            Q.add(G + H)                                          // OR2
```

Fig. 6. Pseudo-code for one-variable decompositions.

$$\begin{aligned} F &= x_i \oplus F_{\overline{x_i}}, && \text{if } \delta F/\delta x_i = 1 \\ F &= \overline{x_i} \oplus F_{x_i}, && \text{if } \delta F/\delta x_i = 1 \\ F &= \overline{x_i} \cdot F_{\overline{x_i}} + F_{x_i}, && \text{if } \exists x_i F = F_{\overline{x_i}} \\ F &= x_i \cdot F_{x_i} + F_{\overline{x_i}}, && \text{if } \exists x_i F = F_{x_i} \end{aligned} \tag{9}$$

The Davio and Shannon expansions are added to the queue in the method described in Fig. 6 (lines 12-14). The simplifications listed in (9) are also checked (lines 4-5, 8 and 10) and preferred to the full Davio and Shannon expansions.

D. Two-variable decompositions

In [32], it is proposed the use of simple cofactor tests in order to perform disjoint-support decompositions. The cofactor tests and decompositions for AND and XOR are described in (10), given $F(X)$ and $\{x, y\} \subseteq X$. These tests are performed in the method of Fig. 7 (lines 4-10, and 18).

If one of the cube-cofactors in (10) is a constant, then simplifications can be derived (lines 14-15 and 20-22). If this is not possible, then a MUX21 gate is defined for the AND decomposition (line 16), and an AX21 gate for the XOR decomposition (line 23).

$$\begin{aligned} F &= \overline{(xy)}F_{\bar{x}} + (xy)F_{xy}, && \text{if } F_{\bar{x}} = F_{\bar{y}} \\ F &= \overline{(x\bar{y})}F_{\bar{x}} + (x\bar{y})F_{x\bar{y}}, && \text{if } F_{\bar{x}} = F_y \\ F &= \overline{(\bar{x}y)}F_x + (\bar{x}y)F_{\bar{x}y}, && \text{if } F_x = F_{\bar{y}} \\ F &= \overline{(\bar{x}\bar{y})}F_x + (\bar{x}\bar{y})F_{\bar{x}\bar{y}}, && \text{if } F_x = F_y \\ F &= ((x \oplus y) \cdot \delta F/\delta x) \oplus F_{\bar{x}\bar{y}}, && \text{if } \delta F/\delta x = \delta F/\delta y \end{aligned} \tag{10}$$

E. Abstraction-based bi-decompositions

A Boolean function F is bi-decomposable if it can be written as $F = G\ op\ H$, where *op* is a Boolean operation and G and H are non-constant functions. This work introduces two methods for bi-decomposition, which are based on the

```
1: procedure DECOMPOSETWOVARIABLES(F, DC, Q)
   Input: ON-set (F), DC-set (DC), priority queue of decompositions Q
   Post: decompositions added to Q
2:   for each pair of variables x_i, x_j ∈ support of F do
3:     found = true                                  // checks AND DSD condition
4:     if F_{x̄_i} == F_{x_j} then
5:       S = (x_i · x_j); G = F_{x_i x_j}; H = F_{x̄_i}
6:     else if F_{x̄_i} == F_{x_j} then
7:       S = (x_i · x̄_j); G = F_{x_i x̄_j}; H = F_{x̄_i}
8:     else if F_{x_i} == F_{x̄_j} then
9:       S = (x̄_i · x_j); G = F_{x̄_i x_j}; H = F_{x_i}
10:    else if F_{x_i} == F_{x_j} then
11:      S = (x̄_i · x̄_j); G = F_{x̄_i x̄_j}; H = F_{x_i}
12:    else found = false
13:    if found then
14:      if G == 0 then Q.add(S̄ · H)                 // AND2
15:      else if G == 1 then Q.add(S + H)            // OR2
16:      else Q.add(S̄ · H + S · G)                   // MUX21
         return
17:    // checks XOR DSD condition
18:    if δF/δx_i == δF/δx_j then
19:      S = (x_i ⊕ x_j), G = F_{x̄ȳ}, H = δF/δx_i
20:      if G == 0 then Q.add(S · H)                 // AND2
21:      else if G == 1 then Q.add(S̄ + H̄)            // OR2
22:      else if H == 1 then Q.add(S ⊕ G)            // XOR2
23:      else Q.add((S · H) ⊕ G)                     // AX21
         return
24:    // two-variable abstraction-based bi-decompositions
25:    if ∃x_i x_j F ≠ 1 then
26:      G = ∃x_i x_j F
27:      H = MINIMIZE(F, Ḡ + DC) // Espresso [28], BDD reduction [29]
28:      Q.add(G · H)                                // AND2
29:    if ∀x_i x_j F ≠ 0 then
30:      G = ∀x_i x_j F
31:      H = MINIMIZE(F, G + DC) // Espresso [28], BDD reduction [29]
32:      Q.add(G + H)                                // OR2
```

Fig. 7. Pseudo-code for two-variable decompositions.

existential and the universal abstractions. As described in [14], these abstractions are related to F as follows:

$$\forall x_i F \leq F \leq \exists x_i F. \tag{11}$$

The existential abstraction $\exists x_i F$ is larger than F. Therefore, it implies an AND decomposition, e.g., $F = \exists x_i F \cdot H$. Similarly, the universal abstraction $\forall x_i F$ is smaller than F, and it implies an OR decomposition, e.g., $F = \forall x_i F + H$. Notice that this method can be applied to any number of variables, as long as the abstractions are not constants, i.e., $\exists x_i F \neq 1$, and $\forall x_i F \neq 0$. In this work, the abstraction-based bi-decompositions are applied to one variable (lines 16-23 in Fig. 6) and two variables (lines 25-32 in Fig. 7).

These abstractions have a characteristic of interest: their support is smaller than F in at least one variable, i.e., $|\exists x_i F| < |F|$, and $|\forall x_i F| < |F|$, given that x_i is in the support of F. Consequently, it is possible to guarantee the support reduction for at least one of the decomposing functions by using these abstractions.

The method proposed differs from other bi-decomposition methods [16]–[18], which try to identify variable partitions and the decomposing functions. The abstraction-based bi-decomposition is applied by setting one of the derived functions (G) to an abstraction ($\exists x_i F$ or $\forall x_i F$), and obtaining the other function (H) via don't care minimization.

The following conditions are used to obtain H via don't care minimization. For the AND bi-decomposition $F = G \cdot H$, $F \leq H \leq F + \overline{G}$, given F and G. Considering $G = \exists x_i F$, then $F \leq H \leq F + \overline{\exists x_i F}$. Similarly, the condition for the OR bi-decomposition $F = G + H$ is $F \cdot \overline{G} \leq H \leq F$, given F and G. Considering $G = \forall x_i F$, then $F \cdot \overline{\forall x_i F} \leq H \leq F$.

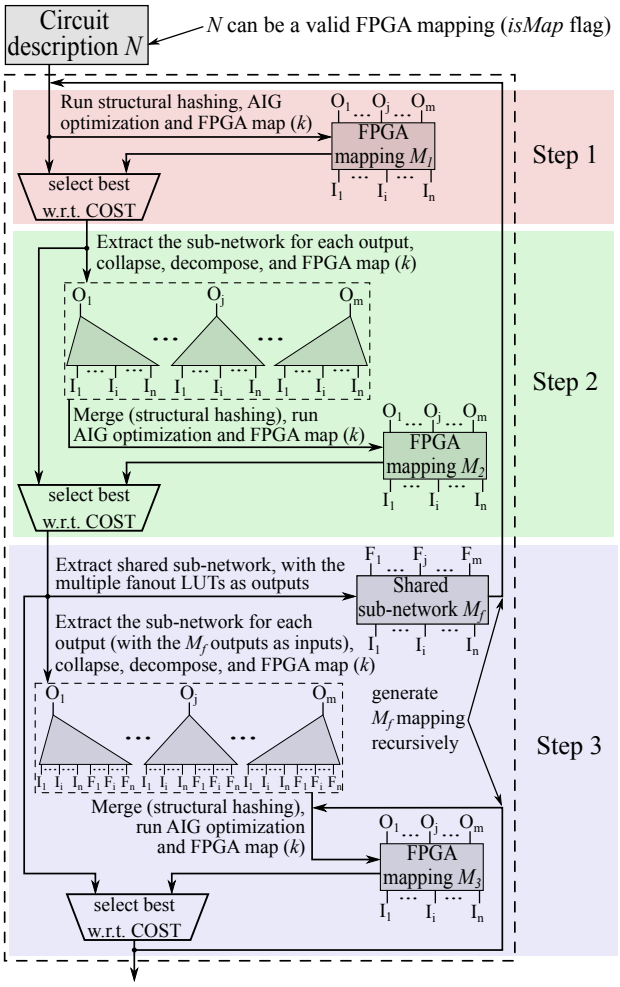


Fig. 8. The recursive remapping approach.

Example: Consider the function $F = abcde\overline{f} + \bar{a}\bar{b}\bar{c}\bar{d}\bar{e}f$. The universal abstraction w.r.t. any variable is the constant 0. Hence, it is not useful to perform the OR bi-decomposition $F = G + H$, since it degenerates to $G = 0$ and $H = F$. On the other hand, the AND bi-decomposition based on the existential abstraction generates a good support-reducing decomposition, as seen in Section II. Consider the existential abstraction w.r.t. variable a: $G = \exists a F = bcde\overline{f} + \bar{b}\bar{c}\bar{d}\bar{e}f$. Using the conditions for H in an AND bi-decomposition ($F \leq H \leq F + \overline{G}$), the following ISF is defined:

$$abcde\overline{f} + \bar{a}\bar{b}\bar{c}\bar{d}\bar{e}f \leq H \leq \bar{a} + \bar{b} + \bar{c} + \bar{d} + \bar{e} + f$$

By applying Boolean minimization, $H = (ab + \bar{a}\bar{b})$ is obtained, and the following AND bi-decomposition is produced:

$$F = (bcde\overline{f} + \bar{b}\bar{c}\bar{d}\bar{e}f)(ab + \bar{a}\bar{b})$$

Notice that this is not a disjoint-support decomposition.

V. Recursive Remapping

This section presents the proposed remapping approach. The idea is to collapse the whole LUT network recursively,

```
1: function COLLAPSEDECOMPOSEMAP(N, k, COST, isMap)
   Input: Circuit description (N), LUT size (k), cost function (COST), flag (isMap)
   Output: An FPGA mapping guided by COST (BestMap)
2:   // Step 1 - obtain M1 by mapping (or remapping) input N
3:   // run structural hashing, AIG optimizations, FPGA mapping
4:   M1 = FPGAMAP(N, k, COST)
5:   // define best mapping result regarding COST
6:   if isMap and COST(N) < COST(M1) then BestMap = N
7:   else BestMap = M1
8:   // if the number of levels is 1, return
9:   if LEVELS(BestMap) == 1 then return BestMap
10:  // Step 2 - obtain M2 by remapping outputs individually
11:  outputNetworks = ∅                          // best mapping for each output
12:  // map each output individually
13:  for each output i in BestMap do
14:      // extract single output network
15:      Ntk_Oi = EXTRACTOUTPUTTOPIS(BestMap, i)
16:      // collapse, decompose, optimize, FPGA mapping
17:      Oi = COLLAPSEFPGAMAP(Ntk_Oi, k, COST)
18:      outputNetworks.insert(Oi, i)
19:  // merge output networks using structural hashing
20:  Ntk_M2 = MERGENETWORKS(outputNetworks)
21:  // run AIG optimizations, FPGA mapping
22:  M2 = FPGAMAP(Ntk_M2, k, COST)
23:  if COST(M2) < COST(BestMap) then BestMap = M2
24:  // Step 3 - obtain M3 by remapping outputs with a shared sub-network
25:  sharedNodes = ∅                            // set of shared nodes used
26:  // create network with all multiple fanout nodes as primary inputs
27:  tempNtk = MULTIPLEFANOUTTOPI(BestMap)
28:  // map each output individually
29:  for each output i in tempNtk do
30:      // extract single output network
31:      Ntk_Oi = EXTRACTOUTPUTTOPIS(tempNtk, i)
32:      // collapse, decompose, optimize, FPGA mapping
33:      Oi = COLLAPSEFPGAMAP(Ntk_Oi, k, COST)
34:      if COST(Oi) < COST(outputNetworks[i]) then
35:          outputNetworks.insert(Oi, i)
36:          sharedNodes.insert(inputs of Oi)
37:  if sharedNodes ≠ ∅ then
38:      // get shared sub-network with sharedNodes as outputs
39:      sharedNtk = GETSHAREDSUBNETWORK(BestMap, sharedNodes)
40:      Mf = COLLAPSEDECOMPOSEMAP(sharedNtk, k, COST, true)
41:      // merge output networks and shared sub-network with struct. hashing
42:      Ntk_M3 = MERGENETWORKS(outputNetworks, Mf)
43:      // run AIG optimizations, FPGA mapping
44:      M3 = FPGAMAP(Ntk_M3, k, COST)
45:      if COST(M3) < COST(BestMap) then BestMap = M3
46:  return BestMap
```

Fig. 9. Pseudo-code of the recursive remapping approach.

decompose, and select the best mapping for each circuit part. An overview of the method is illustrated in Fig. 8, and a pseudo-code detailing the proposed approach is shown in Fig. 9. Different approaches were considered, such as computing maximum fanout-free cones and performing partial collapsing [10]. However, these methods were computationally more expensive and produced worse results than the approach proposed in this section. Notice that windowing and partial collapsing are biased by the structure and by the order that these processes are applied. On the other hand, the recursive remapping proposed is more aggressive, leading to potential (manageable) time-outs, but also larger gains.

The inputs for the remapping approach are a circuit description (N), the number of LUT inputs (k), and a cost function (COST), e.g., area or delay. The description can also be a valid FPGA mapping (for k-LUTs), indicated by the flag *isMap*. The output is an optimized FPGA mapping regarding COST.

The method can be divided into three sequential steps:

Step 1: Obtain LUT network M_1 by mapping (or remapping) the input description N. This is performed in function FPGAMAP (line 4), which runs structural hashing and AIG algebraic optimization. FPGA mapping is performed for each different structure generated, returning the best mapping for the cost function (COST).

Step 2: For each output, extract a single output cone from the LUT network, optimize and map. The mapping is performed in the function COLLAPSEFPGAMAP (line 17), which runs collapsing, decomposition, AIG optimization, and FPGA mapping. Collapsing is a computationally expensive process that may be unfeasible for complex networks, so a time-out is set to avoid a long runtime. If collapsing is successful (and a BDD is obtained, for example), then the decomposition presented in Section IV is applied, generating a *new network*.

If there is a time-out (or if the support size is too large), then the single output network extracted is the only one considered. For each different network, structural hashing is performed, followed by a single execution of AIG optimization scripts and FPGA mapping. The best mapping for each output is greedily selected, and the FPGA mapping M_2 is generated by merging these mappings using structural hashing (line 20), followed by the function FPGAMAP (line 22).

Step 3: Extract a *shared sub-network* from the best mapping found (N, M_1 or M_2). The outputs of this shared sub-network are the nodes with multiple fanout identified in topological order. These nodes are transformed to primary inputs from the outputs perspective (line 27), and the function COLLAPSEFPGAMAP is applied for each output. The implementation for the *sub-network* is obtained recursively (line 40), until the number of levels is 1 (line 9). The FPGA mapping M_3 is generated by merging the output networks and the mapping of the *shared sub-network* using structural hashing (line 42), followed by the function FPGAMAP (line 44). The method returns the best mapping between N, M_1, M_2 and M_3.

The recursive approach creates different optimization opportunities. Regarding M_1, it is possible to derive better solutions by applying AIG optimization in a shared part of the network, instead of the whole circuit. Regarding M_2, if it is not possible to collapse the primary output function, it may be possible for a less complex function, removing part of the structural bias. Furthermore, an area recovery process is achieved, as a common part between outputs is remapped.

VI. EXPERIMENTAL RESULTS

The support-reducing functional decomposition and the recursive remapping are implemented in C++. BDDs are the representation of choice for the functional decomposition, and the CUDD BDD package [30] is used. CUDD provides an intuitive C++ API, and efficient methods to implement the decompositions we propose, with functions to calculate cofactors and abstractions, to identify essential literals, and to perform don't care minimization. Also, BDDs are used in the collapsing of the LUT network, and provide an input for the decomposition without the need of a translation.

The FPGA mapping based on priority cuts [8] and choices [33] implemented in ABC [24] is the one used in the recursive remapping approach. All results passed formal verification with the ABC command *cec*.

TABLE II
FPGA MAPPING COMPARISON WITH BDD-BASED APPROACHES ($k = 5$).

Circuit	BoolMap [9] (delay)		BDS-pga [10] (delay)		ABC (delay)		SR-map (delay) + ABC (delay)		
	LUTs	Lev.	LUTs	Lev.	LUTs	Lev.	LUTs	Lev.	Time(s)
5xp1	**13**	**2**	15	2	21	3	14	2	3
9sym	**7**	**3**	**7**	**3**	60	4	8	3	2
9symml	**7**	**3**	**7**	**3**	58	4	8	3	2
alu2	43	4	41	4	117	7	**35**	**4**	13
alu4	268	7	190	7	219	9	**102**	**4**	26
apex6	188	4	186	4	171	4	**169**	**3**	26
apex7	78	3	71	3	61	3	**54**	**3**	8
b9	41	3	40	3	33	3	**38**	**2**	3
C1355	98	5	**65**	**4**	66	4	66	4	157
C1908	137	7	119	7	95	6	**86**	**6**	217
C499	102	4	**64**	**4**	66	4	66	4	162
C5315	672	9	447	7	365	7	**374**	**6**	217
C880	134	8	108	8	87	7	**92**	**6**	55
clip	**15**	**2**	30	4	71	4	19	3	8
count	**42**	**2**	26	5	36	3	33	3	5
des	**594**	**3**	909	4	623	4	568	4	328
duke2	192	5	169	7	141	4	**120**	**4**	26
misex1	15	2	14	2	15	2	**11**	**2**	2
rd84	**10**	**2**	13	3	109	5	13	3	11
rot	228	6	218	9	203	6	**215**	**5**	43
t481	5	3	**5**	**2**	148	6	**5**	**2**	2
vg2	30	4	**12**	**3**	27	3	21	3	6
z4ml	**5**	**2**	**5**	**2**	**5**	**2**	5	2	1
Geomean	49.63	3.60	44.95	3.91	74.81	4.20	**40.59**	**3.31**	57.52
Ratio	1.00	1.00	0.91	1.09	1.51	1.17	0.82	0.92	-

In order to obtain a *delay-oriented* FPGA mapping with ABC, the command '*if -C 12 -K k*' is used, which primarily targets delay, with a configuration of at most 12 priority cuts per node [25]. Alternatively, *area-oriented* FPGA mapping is obtained with the command '*if -a -C 12 -K k*'. The number of LUT inputs varies for the different sets of benchmarks. Regarding the BDD-based methods, k=5 is defined to compare with the published results. For the remaining cases, the LUT input size is k=6. Structural bias is further reduced by identifying *structural choices* [33] using the commands '*&synch2*' and '*&dch*' on top of the best FPGA mapping obtained.

The proposed approach attempts to optimize a given FPGA mapping. In the experiments presented in this section, the *input* FPGA mapping is either the best known mapping result for the EPFL benchmarks [34], or the FPGA mapping obtained by a commercial tool. For all other cases, the circuit description is used as input, and the mapping is produced by ABC.

The FPGA mappings reported are greedily selected based on the cost function. In this work, two cost functions are analyzed: logic levels and LUT count. If the objective is reducing delay, then SR-map greedily selects the circuit parts with fewer logic levels, using LUT count as a tie breaker. Alternatively, if the goal is to minimize area, LUT count is the main cost function and logic levels is the tie breaker.

A. BDD-based FPGA mapping tools

This section compares the SR-map results with the ones reported by the tools BoolMap [9] and BDS-pga [10], and the mappings obtained with ABC. The FPGA mappings (with k=5) of BoolMap and BDS-pga refer to the best delay results reported in [9] and [10], which are presented in Table II. The bold values in Table II highlight the best delay results. BDS-pga obtains an area reduction of 9% in comparison with BoolMap, at the expense of increasing delay in 9%.

The ABC results are obtained using the same input structure as the BDD-based tools[1], applying structural hashing, 10 iterations of AIG optimization scripts ($compress2rs$ and $dc2$), and delay-oriented FPGA mapping (identifying structural choices). Notice that the number of iterations can be tuned to get better results or to have a lower runtime. The ABC results are the same as the M_1 mappings presented in Section V.

ABC produces worse results than the BDD-based tools, with mappings 51% larger in area and 17% larger in delay, compared to BoolMap. The difference in LUTs is larger than 90% for benchmarks *rd84* and *t481*, and the reason for this behavior lies on the nature of each approach. BoolMap and BDS-pga perform functional transformations using BDDs, whereas ABC performs an structural mapping on top of an AIG, in which nodes and levels are iteratively minimized.

SR-map improves the ABC results and delivers a final result that outperforms BoolMap [9], even using the networks generated by ABC as starting point. This work improves BoolMap results in 18% for area and in 8% for delay, with the best delay result for 12 of the 22 benchmarks.

B. 20 largest MCNC benchmarks

This section presents results for the 20 largest MCNC benchmarks, comparing the results of this work with the ones obtained with a commercial tool and ABC. The synthesis in the commercial tool is configured to avoid the use of multiplexers and merging of LUTs, delivering results comparable to the other tools. The reported runtime for the commercial tool regards only the logic synthesis and optimization steps.

Table III presents the FPGA mappings to LUTs with k=6. The bold numbers in '*Levels*' highlight the best delay results, whereas the bold numbers in '*LUTs*' underline the best results for area. Regarding the methods analyzed, SR-map obtains the best delay result for 19 of 20, and the best area result for 13 of the 20 benchmarks. All results are generated with the same input description. The ABC mapping is obtained with structural hashing, 10 iterations of AIG optimization scripts, and FPGA mapping identifying structural choices.

BDS-pga [10] and MFS [25] results are ommited due to space, but their relationship is ($>$ means better results): commercial tool $>$ MFS [25] $>$ BDS-pga [10] $>$ ABC (AIG optimization, and FPGA mapping with priority cuts and structural choices). Notice that ABC results are the starting point for the proposed method, showing the difference of using the same mapping algorithm but exploring different structures.

1) Delay-oriented mapping: Using the delay-oriented FPGA mapping, ABC produces a result 99% larger in area and 5% larger in delay when compared with the commercial tool. SR-map produces a result with 27% fewer logic levels and 10% fewer LUTs, with ABC delay-oriented mapping, and delay as cost function (COST). Also, an area reduction of 16% plus a delay reduction of 12% is achieved when area is defined as the cost function in SR-map.

[1] http://www.ecs.umass.edu/ece/tessier/rcg/bds-pga-2.0/blifs.tar

TABLE III
FPGA MAPPING COMPARISON FOR THE 20 LARGEST MCNC BENCHMARKS ($k = 6$).

								Recursive remapping										
Circuit	Commercial tool			ABC (delay)		ABC (area)		Factor (delay) + ABC (delay)		Factor (area) + ABC (delay)		SR-map (delay) + ABC (delay)			SR-map (area) + ABC (delay)			SR-map (area) + ABC (area)	
	LUTs	Lev.	Time(s)	LUTs	Lev.	LUTs	Lev.	LUTs	Lev.	LUTs	Lev.	LUTs	Lev.	Time(s)	LUTs	Lev.	Time(s)	LUTs	Lev.
alu4	320	5	230	456	5	415	10	183	5	186	5	63	**3**	24	58	4	21	**56**	5
apex2	302	13	276	507	6	408	11	40	4	40	4	35	**3**	17	34	4	14	**30**	7
apex4	192	3	290	558	5	529	10	390	4	390	4	155	3	203	155	3	212	**153**	**3**
bigkey	569	3	313	577	3	577	3	577	3	577	3	685	**2**	257	**491**	3	191	577	3
clma	**180**	5	438	2614	8	2221	18	200	4	203	4	203	4	75	190	**4**	84	186	8
des	**436**	4	345	447	4	457	7	447	4	446	4	513	**3**	194	445	4	222	449	5
diffeq	**472**	8	348	559	7	510	13	559	7	532	8	533	**7**	241	527	8	262	502	14
dsip	**690**	3	338	871	3	871	3	869	**2**	869	**2**	869	**2**	190	869	**2**	224	869	**2**
elliptic	**115**	**5**	313	315	6	297	11	315	6	315	6	316	5	36	311	6	51	291	11
ex1010	210	3	335	572	5	550	10	453	4	432	5	208	**3**	107	**207**	4	180	**207**	4
ex5p	100	2	252	326	4	301	9	91	2	85	3	86	2	12	**82**	**2**	16	82	5
frisc	1694	13	292	1725	12	1698	26	1886	10	1692	13	1857	**10**	654	1689	13	853	**1637**	25
i10	557	9	285	535	8	500	22	627	7	522	9	537	**7**	329	505	9	317	**481**	24
misex3	197	5	240	284	5	234	9	205	4	193	4	117	4	37	102	**4**	35	**94**	5
pdc	155	4	286	1385	6	1143	14	154	4	148	4	157	**3**	70	**142**	4	67	144	6
s38417	**1458**	7	437	2557	6	2443	11	2460	6	2458	7	2450	**6**	1304	2396	7	1035	2369	12
s38584	**1946**	8	432	2287	6	2255	12	2463	5	2212	6	2224	**5**	1246	2229	5	729	2198	12
seq	531	7	247	583	5	520	10	560	4	486	5	527	**4**	307	459	5	114	**420**	10
spla	157	4	269	1350	6	1128	15	145	4	137	4	156	**3**	138	135	4	62	**121**	6
tseng	656	8	269	651	6	631	13	647	6	635	8	634	**6**	385	636	8	265	**629**	12
Geomean	374.30	5.24	306.1	744.07	5.51	685.86	10.56	400.08	4.43	383.36	4.92	335.79	**3.84**	141.1	312.67	4.61	128.9	**304.62**	7.21
Ratio	1.00	1.00	-	1.99	1.05	1.83	2.01	1.07	0.84	1.02	0.94	0.90	**0.73**	-	0.84	0.88	-	**0.81**	1.38

2) Area-oriented mapping: ABC produces a result 83% larger in area than the commercial tool using area-oriented FPGA mapping. Notice that there is an area recovery post-process in ABC delay-oriented mapping, but area-oriented mapping does not try to improve delay. Therefore, delay is increased significantly, almost doubling the logic levels. Using ABC area-oriented mapping and area as cost function, SR-map obtains a result with 19% fewer LUTs than the commercial tool, but with 38% more logic levels. The results are 3% smaller in area than using ABC delay-oriented mapping, but with much worse delay results, as this is disregarded in ABC area-oriented mapping. For this reason, this configuration is not recommended if delay must be considered.

3) Support-reducing decomposition: The remapping approach proposed in Section V can be applied regardless of the support-reducing decomposition presented in Section IV. For example, the collapsed functions can be decomposed using algebraic factorization [28], instead of the method proposed. The results for the recursive remapping using factorization (obtained with the ABC command *strash*) instead of the support-reducing decomposition are also presented in Table III, denoted as '*Factor*'. For some benchmarks, e.g., *apex2*, *clma*, *ex5p*, the removal of the structural bias using recursive remapping produces similar results both for factorization and decomposition. However, considering the full set of benchmarks, the results obtained using the support-reducing decomposition are considerably better than the ones using factorization, both for area and delay.

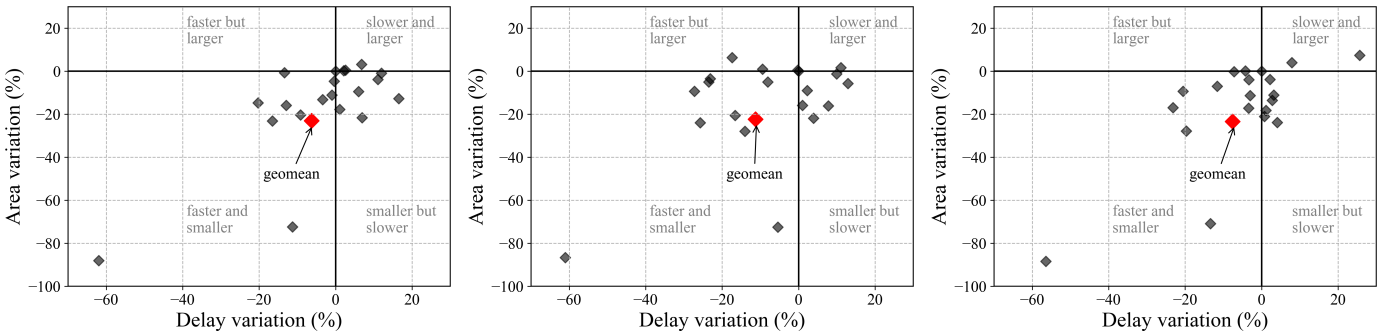
C. EPFL benchmarks

The EPFL benchmarks [34] are a set of 20 designs, 10 arithmetic and 10 random/control circuits. Since 2015, the best known FPGA mapping results (with k=6) for delay and for area are recorded. Consequently, these benchmarks have highly optimized results, which are very difficult to improve. For example, the commercial tool used in this work is not able to improve any of the EPFL results, as it provides FPGA mappings with more balanced results in area and delay, and also considers congestion issues.

TABLE IV
BEST KNOWN RESULTS FOR EPFL BENCHMARKS (2017).

Circuit	Best EPFL (delay)		SR-map (delay) + ABC (delay)		
	LUTs	Levels	LUTs	Levels	Time(s)
arbiter	2884	5	2243	5	729
cavlc	115	4	75	3	25
dec	270	2	264	2	11
int2float	41	3	31	3	5
i2c	244	3	242	3	30
mem_ctrl	2490	7	2484	6	792
max	882	10	857	10	313
multiplier	8215	28	6543	28	26012
priority	157	4	152	4	15
router	57	4	54	4	5
sin	1801	30	3546	28	12779
voter	1469	12	1450	12	4601
Circuit	Best EPFL (area)		SR-map (area) + ABC (area)		
	LUTs	Levels	LUTs	Levels	Time(s)
cavlc	101	6	72	4	25
dec	270	2	264	2	11
int2float	28	6	27	6	5

The proposed method is able to update 12 of the best known results for delay, and 3 for area, as presented in Table IV. The most remarkable results are: *cavlc*, with a reduction of 25% in delay plus 35% in area; *int2float*, reducing LUT count in 25%; and the *multiplier*, with an area reduction of 20%. Note that the area of the *sin* benchmark is increased significantly for a reduction of 2 logic levels. This behavior is expected, as the delay is the cost function in this case. Therefore, SR-map greedily selects the circuit parts with lowest logic levels, considering LUT count only as a tie breaker.



(a) Default strategy (b) Area strategy (c) Delay strategy

Fig. 10. Comparison of the commercial tool results using the SR-map as input vs. the initial description, for the synthesis strategies: Default, Area, Delay.

TABLE V
REMAPPING OF THE COMMERCIAL TOOL RESULTS FOR THE 20 LARGEST MCNC BENCHMARKS ($k = 6$).

Circuit	ABC (delay)		SR-map (delay) + ABC (delay)			SR-map (area) + ABC (delay)		
	LUTs	Levels	LUTs	Levels	Time(s)	LUTs	Levels	Time(s)
alu4	232	5	65	3	28	58	4	19
apex2	179	6	33	4	13	32	4	9
apex4	193	3	155	3	107	155	3	86
bigkey	573	3	681	2	285	459	3	144
clma	193	3	181	3	24	166	4	17
des	446	4	445	3	232	436	4	178
diffeq	470	6	470	6	261	452	7	223
dsip	869	2	869	2	215	681	2	181
elliptic	105	4	110	3	21	99	4	16
ex1010	210	3	208	3	96	207	4	76
ex5p	92	2	86	2	10	82	2	7
frisc	1839	10	1892	9	947	1684	11	792
i10	551	8	533	8	302	512	9	280
misex3	158	4	112	4	28	96	4	21
pdc	126	4	148	3	23	114	4	15
s38417	2493	6	2432	6	967	1458	7	971
s38584	2188	6	2332	5	640	1946	8	641
seq	458	5	417	5	89	446	5	77
spla	131	4	145	3	20	119	4	15
tseng	649	6	636	6	240	625	7	237
Geomean	360.88	4.32	306.47	3.76	94.8	270.47	4.52	74.3
Ratio	0.96	0.82	0.82	0.72	-	0.72	0.86	-

D. Remapping of the results from a commercial tool

The results in Table III are obtained from the original BLIF descriptions. In this section, the results obtained with the commercial tool are remapped by SR-map. The remapping results are presented in Table V. The commercial tool performs sequential optimization, and equivalence checking with the original description is performed with ABC command *dsec*.

A reduction of 4% in area and 18% in delay of the results from a commercial tool is obtained with ABC by performing iterative AIG transformations and FPGA mapping with choices. Using the ABC delay-oriented mapping and delay as cost function, SR-map achieves even better results, with 28% fewer logic levels and 18% fewer LUTs. Also, an area reduction of 28% plus a delay reduction of 14% is obtained when area is defined as the cost function for SR-map.

E. SR-map remapping as input to the commercial tool

Previous experiments are presented with results in number of LUTs and levels, but reduction in logic levels often does not translate into improved delay post place-and-route due to congestion issues. In order to evaluate this effect, SR-map results are fed back to the commercial tool, comparing the post place-and-route metrics between using the initial description versus using the SR-map remapping as input.

TABLE VI
RESULTS OF A COMMERCIAL TOOL FOR DIFFERENT STRATEGIES, AFTER PHYSICAL SYNTHESIS (PLACE-AND-ROUTE).

Input from:	Synthesis strategy of the tool					
	Default		*Area*		*Delay*	
	Area	Delay	Area	Delay	Area	Delay
Initial description	330.73	4.94	320.32	5.65	375.24	4.92
SR-map (delay) + ABC(delay)	-20%	-9%	-16%	-9%	-20%	-6%
SR-map (area) + ABC(delay)	-23%	-6%	-22%	-11%	-23%	-8%

A summary of the geometric mean results is presented in Table VI. Three different synthesis strategies are investigated: (1) with the *Default* parameters, (2) targeting *Area* minimization, and (3) *Delay* reduction. The delay reported is the one for the critical path, in ns.

In previous experiments, the synthesis of the commercial tool was configured to avoid the use of multiplexers and merging of LUTs, delivering results comparable to the other tools. In this section, the results presented may have multiplexers, and the LUTs may be merged. Additionally, sequential optimization is performed, and the number of registers may vary for the different synthesis strategies and inputs. The area reported considers these characteristics of the commercial tool: logic optimization and merging (LUTs), sequential optimization (FFs), and the use of multiplexers to reduce delay and implement functions with more inputs (7 and 8). Multiplexers occupy much less area than LUTs and flip-flops, so we conservatively consider muxes as half the size of the other elements, resulting in this function to compare area: $Area = LUTs + FFs + 0.5 \times Multiplexers$.

In comparison with the *Default strategy*, and using the initial description, the *Area strategy* improves 3% in area for an increase of 14% in delay, whereas the *Delay strategy* reduces delay by 1% at the expense of increasing area by 14%.

As observed in Table VI, the post place-and-route results show worse metrics than the ones reported in Table V, as these are implemented into an actual FPGA. Still, for the *Default strategy*, the results have 20% less area and 9% less delay, by

using the SR-map remapping targeting delay as input, and 23% less area and 6% less delay, by using the SR-map remapping targeting area as input. Similar results are achieved for the other strategies. Notice that the commercial tool results in Table III are for a different synthesis strategy, in which the use of multiplexers and LUT combining is avoided.

Interestingly, the SR-map remapping targeting area as input to the commercial tool generated better post place-and-route results, even for delay. This is because aggressive reduction of logic levels often leads to congestion, which prevents the place-and-route tools to translate this reduction into better performance. A detailed comparison of the benchmark results for this case is provided in the plots of Fig. 10.

F. Scalability analysis

Boolean methods are known to have scalability issues, and this is typically handled by limiting the scope of application, with techniques such as partial collapsing [9]–[11], windowing [25], and partitioning [35]. The proposed approaches are no different. The size of BDDs may increase significantly with the number of variables, slowing down even simple BDD operations. For this reason, the collapsing process is only applied to functions with up to a certain limit of input variables. Fig. 11 shows the average area-delay product for different limits in the support, presenting a trade-off between runtime and quality of results. The results presented are obtained with SR-map for a subset of the benchmarks considered in previous sections. For the results previously presented in this paper, the support limit is defined to 50 variables.

Additionally, a time-out is set to prevent a long runtime. This runtime limit is set for the collapsing process and the decomposition. Notice that the BDD size may be large and the BDD operations may take a long time, even with a limit in the number of variables. As observed in Fig. 11, the quality of results obtained by SR-map is similar for a time-out of 5 and 1000 seconds. The runtime with time-out of 1000 seconds exposes the exponential behavior of increasing the number of variables for BDD operations. However, the runtime can be kept under control by defining a smaller time-out, which also presents a linear increase with the support limit. As mentioned in Section V, if there is a time-out (or if the support size is larger than the limit defined), then the output network extracted is the only one considered in the remapping algorithm.

Also, note that the remapping method extracts a sub-network that feeds all outputs, therefore a larger number of levels in the FPGA mapping may also result in a larger runtime. The ABC tool presents a fraction of the runtime obtained with SR-map, as it is part of the proposed method, and it is repeatedly used. Nevertheless, the average execution time observed in Table III (141 and 129 seconds) is comparable with the one for the commercial tool (306 seconds).

VII. Conclusions

This paper proposes a support-reducing functional decomposition method to produce a subject graph with a structure more suitable to LUT-based FPGA mapping. An recursive remapping approach is also proposed, trying to reduce the structural bias of the circuit, and using the actual FPGA mapping result as cost function.

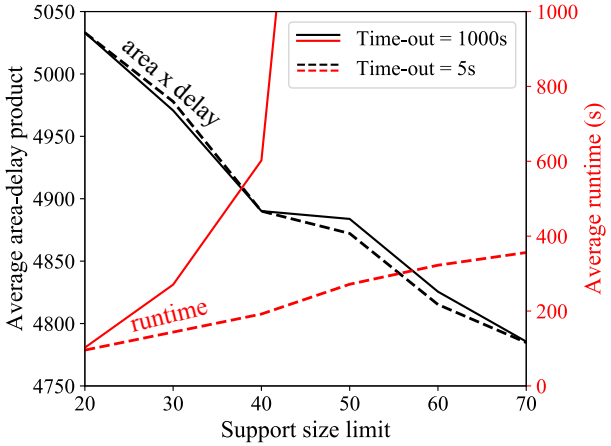


Fig. 11. Runtime and quality analysis, considering different limits for the support (20 to 70 variables), and time-outs (5 and 1000 seconds). The area-delay product is the result of number of LUTs times the number of levels.

The experiments show promising results. The proposed method improves the FPGA mapping results of a commercial tool for the MCNC benchmarks, with gains of 28% in delay plus 18% in area when targeting delay, and 28% in area plus 14% in delay with area as cost function. Post place-and-route results with 23% less area and 6% less delay (or 20% less area and 9% less delay) are obtained by using the remapping result as input to the commercial tool instead of the initial description. Moreover, 12 of the best known results for delay (and 3 for area) of the EPFL benchmarks are updated.

As future work, some directions could be explored. Additional support-reducing techniques could be incorporated, such as [16], [18]. A partial collapsing approach that uses the FPGA mapping result as cost function could be investigated. Regarding the recursive remapping, the propagation of the don't care conditions could potentially improve the results. Also, keeping track of the critical paths may allow further area reduction while obtaining similar delay results.

References

[1] L. Machado and J. Cortadella, "Support-reducing functional decomposition for FPGA technology mapping," in *Proc. of the International Workshop on Logic and Synthesis*, 2018, pp. 79–86.

[2] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 203–215, 2007.

[3] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 13–24, 2014.

[4] Y.-k. Choi, J. Cong, Z. Fang, Y. Hao, G. Reinman, and P. Wei, "A quantitative analysis on microarchitectures of modern CPU-FPGA platforms," in *Proc. of the Design Automation Conference*. ACM, 2016, p. 109.

[5] G. Liu and Z. Zhang, "A parallelized iterative improvement approach to area optimization for LUT-based technology mapping," in *Proc. of the International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 147–156.

[6] J. Cong and Y. Ding, "FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 1, pp. 1–12, 1994.

[7] D. Chen and J. Cong, "DAOmap: A depth-optimal area optimization mapping algorithm for FPGA designs," in *Proc. of the International Conference on Computer-Aided Design*. IEEE, 2004, pp. 752–759.

[8] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton, "Combinational and sequential mapping with priority cuts," in *Proc. of the International Conference on Computer-Aided Design*. IEEE, 2007, pp. 354–361.

[9] C. Legl, B. Wurth, and K. Eckl, "A Boolean approach to performance-directed technology mapping for LUT-based FPGA designs," in *Proc. of the Design Automation Conference*. ACM, 1996, pp. 730–733.

[10] N. Vemuri, P. Kalla, and R. Tessier, "BDD-based logic synthesis for LUT-based FPGAs," *ACM Transactions on Design Automation of Electronic Systems*, vol. 7, no. 4, pp. 501–525, 2002.

[11] L. Cheng, D. Chen, and M. D. Wong, "DDBDD: Delay-driven BDD synthesis for FPGAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1203–1213, 2008.

[12] M. Kubica, A. Opara, and D. Kania, "Logic synthesis for FPGAs based on cutting of BDD," *Microprocessors and Microsystems*, vol. 52, pp. 173–187, 2017.

[13] H. Sawada, T. Suyama, and A. Nagoya, "Logic synthesis for look-up table based FPGAs using functional decomposition and support minimization," in *Proc. of the IEEE/ACM international conference on Computer-aided design*. IEEE, 1995, pp. 353–358.

[14] F. M. Brown, *Boolean reasoning: the logic of Boolean equations*. New York, NY, USA: Springer, 2012.

[15] C. Legl, B. Wurth, and K. Eckl, "An implicit algorithm for support minimization during functional decomposition," in *Proc. of European Design and Test Conference*. IEEE, 1996, pp. 412–417.

[16] A. Mishchenko, B. Steinbach, and M. Perkowski, "An algorithm for bi-decomposition of logic functions," in *Proc. of the Design Automation Conference*. ACM, 2001, pp. 103–108.

[17] R.-R. Lee, J.-H. R. Jiang, and W.-L. Hung, "Bi-decomposing large Boolean functions via interpolation and satisfiability solving," in *Proc. of the 45th annual Design Automation Conference*. ACM, 2008, pp. 636–641.

[18] M. Choudhury and K. Mohanram, "Bi-decomposition of large Boolean functions using blocking edge graphs," in *Proc. of the International Conference on Computer-Aided Design*. IEEE, 2010, pp. 586–591.

[19] V. N. Kravets and K. A. Sakallah, "Constructive library-aware synthesis using symmetries," in *Proc. of Design, Automation and Test in Europe*. ACM, 2000, pp. 208–215.

[20] A. Mishchenko and R. Brayton, "Scalable logic synthesis using a simple circuit structure," in *Proc. of the International Workshop on Logic and Synthesis*, 2006, pp. 15–22.

[21] J. Cortadella, "Timing-driven logic bi-decomposition," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 22, no. 6, pp. 675–685, 2003.

[22] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting a fresh look at combinational logic synthesis," in *Proc. of the Design Automation Conference*. ACM, 2006, pp. 532–535.

[23] R. Brayton, J.-H. R. Jiang, and S. Jang, "SAT-based logic optimization and resynthesis," in *Proc. of the International Workshop on Logic and Synthesis*, 2007, pp. 358–364.

[24] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Proc. of the International Conference on Computer-Aided Verification*, 2010, pp. 24–40.

[25] A. Mishchenko, R. Brayton, J. R. Jiang, and S. Jang, "Scalable don't-care-based logic optimization and resynthesis," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 4, no. 4, p. 34, 2011.

[26] P. Fišer, J. Schmidt, and J. Balcárek, "Sources of bias in EDA tools and its influence," in *Proc. of the International Symposium on Design and Diagnostics of Electronic Circuits & Systems*, 2014, pp. 258–261.

[27] M. Damiani and G. De Micheli, "Don't care set specifications in combinational and synchronous logic circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, no. 3, pp. 365–388, 1993.

[28] R. K. Brayton, A. L. Sangiovanni-Vincentelli, C. T. McMullen, and G. D. Hachtel, *Logic minimization algorithms for VLSI synthesis*. Norwell, MA, USA: Kluwer, 1984.

[29] Y. Hong, P. A. Beerel, J. R. Burch, and K. L. McMillan, "Safe BDD minimization using don't cares," in *Proc. of the Design Automation Conference*. ACM, 1997, pp. 208–213.

[30] F. Somenzi, "CUDD: CU decision diagram package release 3.0.0," *University of Colorado at Boulder*, 2015.

[31] D. Bañeres, J. Cortadella, and M. Kishinevsky, "Timing-driven N-way decomposition," in *Proc. of the Great Lakes Symposium on VLSI*, 2009, pp. 363–368.

[32] V. Callegaro, F. S. Marranghello, M. G. Martins, R. P. Ribas, and A. I. Reis, "Bottom-up disjoint-support decomposition based on cofactor and Boolean difference analysis," in *Proc. of the IEEE International Conference on Computer Design*. IEEE, 2015, pp. 680–687.

[33] S. Chatterjee, A. Mishchenko, R. K. Brayton, X. Wang, and T. Kam, "Reducing structural bias in technology mapping," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 12, pp. 2894–2903, 2006.

[34] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "The EPFL combinational benchmark suite," in *Proc. of the International Workshop on Logic & Synthesis*, 2015, pp. 57–61.

[35] M. Fujita, Y. Matsunaga, Y. Tamiya, and K.-C. Chen, "Multi-level logic minimization of large combinational circuits by partitioning," in *Logic Synthesis and Optimization*. Boston, MA, USA: Springer, 1993, pp. 109–126.

Lucas Machado received the B.S. and the M.S. degrees in computer engineering from the Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil, in 2010 and 2013, respectively. He is currently pursuing the Ph.D. degree in computer science at the Universitat Politècnica de Catalunya, Barcelona, Spain. His current research interests include computer-aided design of integrated circuits, with special interest on logic synthesis, reliability, security, and asynchronous circuits.

Jordi Cortadella (S'87-M'89-F'15) is a Professor with the Computer Science Department, Universitat Politècnica de Catalunya, Barcelona, Spain. He holds an M.S. and a Ph.D. degree in Computer Science (Universitat Politècnica de Catalunya, 1985 and 1987). His current research interests include formal methods and computer-aided design of VLSI systems with a special emphasis on asynchronous circuits, concurrent systems, and logic synthesis. Prof. Cortadella is a member of Academia Europaea. He received best paper awards at the International Symposium on Advanced Research in Asynchronous Circuits and Systems in 2004 and 2016, the Design Automation Conference in 2004, and the International Conference on Application of Concurrency to System Design in 2009. He has served on the technical committees of several international conferences in the field of design automation and concurrent systems, and is an Associate Editor of the *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.